EECS 280 – Lecture 3

Pointers

5/5/2022

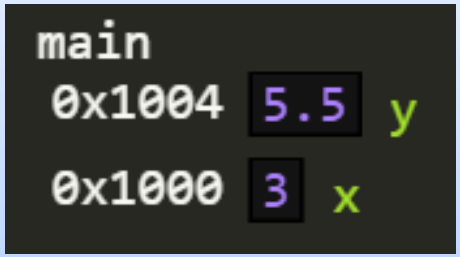
C++ Memory Model

- An **object** is a piece of data in memory.
- An object lives at an **address** in memory.
- You can use an object during its **lifetime**.
- Lifetimes are managed according to **storage duration**. Three options in C++:
  - ***Static***
    Lives for the whole program.
  - ***Automatic (Local)***
    Lives during the execution of its local block.
  - ***Dynamic***
    You control the lifetime!

Managed by the compiler.

Addresses

- Every object lives at some **address** in memory.
  - This is determined by the compiler. You really don't have any control over it.
- You can get the address of an object using the & operator.

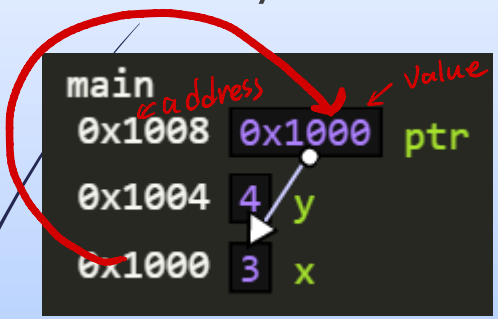


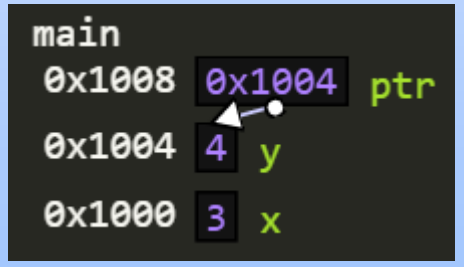

```
int main() {
  int x = 3;
  double y = 5.5;
  cout << &x << endl; // prints 0x1000
  cout << &y << endl; // prints 0x1004
}
```

hexadecimal notation

Pointers

- We can also create objects to store addresses. These are called **pointers**.[1]
- To declare a pointer variable, affix the * symbol to the left of the name.






```
int main() {
  int x = 3;
  int y = 4;
  int *ptr = &x;
  cout << ptr << endl; // prints 0x1000
  ptr = &y; // assign a new address to ptr
  cout << ptr << endl; // prints 0x1004
}
```

1 The terms "address" and "pointer" are often used interchangeably, though technically a pointer holds an address as its value.

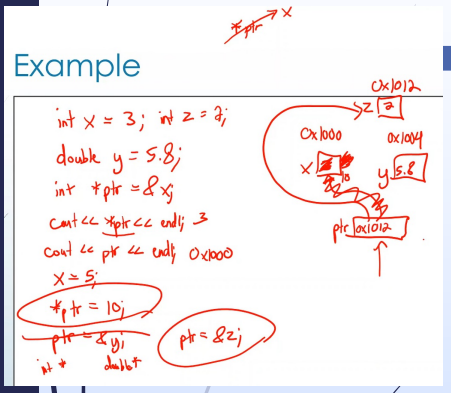
Pointers

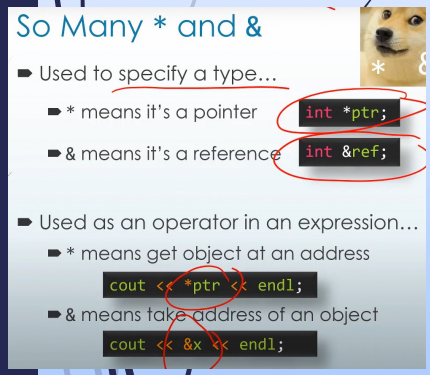
- There is a separate pointer type for each kind of thing you could point to, and you can't mix them.

double pointer : int* *d_ptr = &ptr1;

```
int main() {
  int x = 3;
  double y = 4;
  int *ptr1 = &x;
  double *ptr2 = &y;
}
```

Using Pointers in Expressions





- To take the address of an object, use the & operator.
  - Pronounced as "address of".
  - Yields a pointer.
- To get the object a pointer points to, use the * operator.
  - Pronounced as "star", "dereference", or "indirection".
  - "Follows" the pointer to its object.

Minute Exercise

```
int main() {
  int foo = 1;
  int *bar = &foo;
  foo = 2;
  *bar = 3;

  cout << "foo = " << foo << endl;
  cout << "bar = " << bar << endl;
  cout << "*bar = " << *bar << endl;
}
```

Question

How many times does the object at 0x2710 change value (not including when it is initialized)?

A) 1
B) 2
C) 3
D) 4

Exercise 1

- Exercise L03.1_pointers at lobster.eecs.umich.edu
  - Don't run the code right away!

https://bit.ly/38Xog37

Null and Uninitialized Pointers

Picture

Null and Uninitialized Pointers

- A **null pointer** has value `0x0` (i.e. it points to address 0)
  - No objects are allowed to live at address 0.
  - A null pointer is interpreted as "not pointing to anything".
  - Dereference a null pointer → **runtime error** (usually).
  - Declare a null pointer like this: `int *ptr = nullptr;`
- Just like any other variable of primitive type, an uninitialized pointer has no value in particular.
  - It's pointing at some random place in memory!
  - Dereference an uninitialized pointer → **undefined behavior**.
    - Maybe it crashes? If the pointer is pointing to memory your program isn't allowed to use, you might get a **segmentation fault**.
    - Maybe you read some random memory and get junk values?
    - Maybe you write some random memory and mess up other stuff.

Exercise 2

https://bit.ly/3OZ6RHL

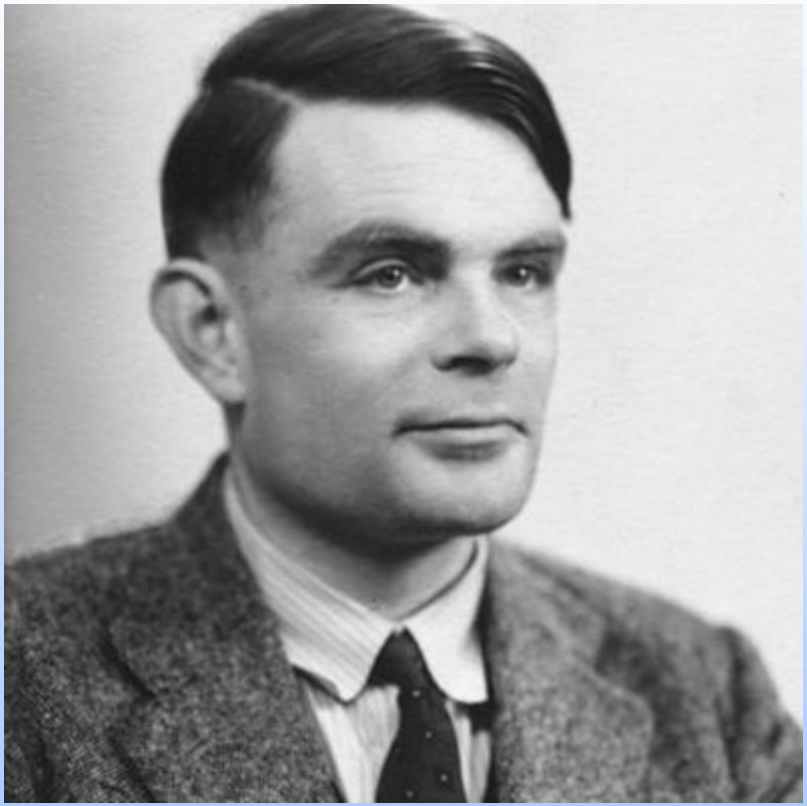
Break Time!

https://xkcd.com/138/

Break Time!

https://xkcd.com/371/

Alan Turing



Computer Scientist and Mathematician

Foundational work in theoretical computer science and artificial intelligence

Ada Lovelace

The First Computer Programmer

Diagram for the computation by the Engine of the N

Number of Operation.	Nature of Operation.	Variables acted upon.	Variables receiving results.	Indication of change in the value on any Variable.	Statement of Results.	Data.						
						$^1V_1$ ○ 0 0 1 [1]	$^1V_2$ ○ 0 0 2 [2]	$^1V_3$ ○ 0 0 4 [n]	$^0V_4$ ○ 0 0 0 [ ]	$^0V_5$ ○ 0 0 0 [ ]	$^0V_6$ ○ 0 0 0 [ ]	
1	×	$^1V_2 \times {}^1V_3$	$^1V_4, {}^1V_5, {}^1V_6$	$\left\{ \begin{matrix} {}^1V_2 = {}^1V_2 \\ {}^1V_3 = {}^1V_3 \end{matrix} \right\}$	$= 2n$	...	2	n	$2n$	$2n$	$2n$	
2	−	$^1V_4 - {}^1V_1$	$^2V_4$	$\left\{ \begin{matrix} {}^1V_4 = {}^2V_4 \\ {}^1V_1 = {}^1V_1 \end{matrix} \right\}$	$= 2n-1$	1	...	...	$2n-1$			
3	+	$^1V_5 + {}^1V_1$	$^2V_5$	$\left\{ \begin{matrix} {}^1V_5 = {}^2V_5 \\ {}^1V_1 = {}^1V_1 \end{matrix} \right\}$	$= 2n+1$	1	...	...	...	$2n+1$		
4	÷	$^2V_5 \div {}^2V_4$	$^1V_{11}$	$\left\{ \begin{matrix} {}^2V_5 = {}^0V_5 \\ {}^2V_4 = {}^0V_4 \end{matrix} \right\}$	$= \frac{2n-1}{2n+1}$	...	...	...	0	0	...	
5	÷	$^1V_{11} \div {}^1V_2$	$^2V_{11}$	$\left\{ \begin{matrix} {}^1V_{11} = {}^2V_{11} \\ {}^1V_2 = {}^1V_2 \end{matrix} \right\}$	$= \frac{1}{2} \cdot \frac{2n-1}{2n+1}$	...	2	...	...	...	...	
6	−	$^0V_{13} - {}^2V_{11}$	$^1V_{13}$	$\left\{ \begin{matrix} {}^2V_{11} = {}^0V_{11} \\ {}^0V_{13} = {}^1V_{13} \end{matrix} \right\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$	...	...	...	...	...	...	
7	−	$^1V_3 - {}^1V_1$	$^1V_{10}$	$\left\{ \begin{matrix} {}^1V_3 = {}^1V_3 \\ {}^1V_1 = {}^1V_1 \end{matrix} \right\}$	$= n-1\,(=3)$	1	...	n	...	...	...	
8	+	$^1V_2 + {}^0V_7$	$^1V_7$	$\left\{ \begin{matrix} {}^1V_2 = {}^1V_2 \\ {}^0V_7 = {}^1V_7 \end{matrix} \right\}$	$= 2+0=2$	...	2	...	...	...	...	
9	÷	$^1V_6 \div {}^1V_7$	$^3V_{11}$	$\left\{ \begin{matrix} {}^1V_6 = {}^1V_6 \\ {}^0V_{11} = {}^3V_{11} \end{matrix} \right\}$	$= \frac{2n}{2} = A_1$	...	...	...	...	...	$2n$	
10	×	$^1V_{21} \times {}^3V_{11}$	$^1V_{12}$	$\left\{ \begin{matrix} {}^1V_{21} = {}^1V_{21} \\ {}^3V_{11} = {}^3V_{11} \end{matrix} \right\}$	$= B_1 \cdot \frac{2n}{2} = B_1 A_1$	...	...	...	...	...	...	
11	+	$^1V_{12} + {}^1V_{13}$	$^2V_{13}$	$\left\{ \begin{matrix} {}^1V_{12} = {}^0V_{12} \\ {}^1V_{13} = {}^2V_{13} \end{matrix} \right\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}$ ...	...	...	...	...	...	...	
12	−	$^1V_{10} - {}^1V_1$	$^2V_{10}$	$\left\{ \begin{matrix} {}^1V_{10} = {}^2V_{10} \\ {}^1V_{\cdot} = {}^1V_{\cdot} \end{matrix} \right\}$	$= n-2\,(=2)$	1	...	...	...	...	...	

A diagram by Lovelace contains the first recorded algorithm for implementation by a computer.

Break Time!

We'll start again in one minute.

Pass-by-Pointer

```
void addOne(int *x) {
  *x += 1; // adds 1 to whatever x points to, which
           // is z in this example, even though the
           // name z is not in scope here.
           // We used the address of z to get to it.
}

int main() {
  int z = 1;
  int *ptr = &z; // ptr "points to" z

  *ptr += 1; // adds 1 to z, without using the name z

  addOne(&z); // adds one to z
  addOne(ptr); // same thing
}
```

Note that pass-by-pointer is really just pass-by-value, which makes a copy. But a copy of the address still lets you get back to the original object!

Exercise: Swap with Pointers

- `L03.3_pointer_swap` on Lobster.

```
void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 5;
  swap(a, b);
  cout << "a = " << a << endl;
  cout << "b = " << b << endl ;
}
```

This code is broken! The swap function does nothing. Fix it by changing the parameters to use pass-by-pointer!

Solution: Swap with Pointers

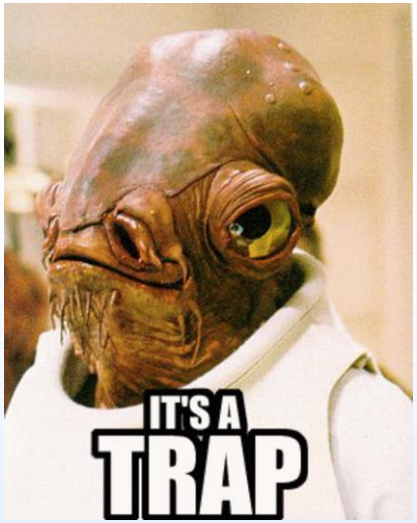
- L03.3_pointer_swap on Lobster.

```
void swap(                    ) {




}

int main() {
  int a = 3;
  int b = 5;
  swap(a, b);
  cout << "a = " << a << endl;
  cout << "b = " << b << endl ;
}
```

Picture

Exercise: Pointer Trap

- `L03.4_pointers` on Lobster.




```
int * getAddress(int x) {
  return &x; // It's a trap!
}
void printAnInt(int anInt) {
  cout << anInt << endl;
}
int main() {
  int a = 3;
  int *ptr = getAddress(a);

  printAnInt(42);

  // should print 3, right???
  cout << *ptr << endl;
}
```

Clicker Question

Why is it a trap?

A) Can't return pointers from functions

B) `anInt` became a reference to x

C) The lifetime of the parameter x ended before `*ptr` was used

D) `ptr` became uninitialized when `printAnInt` was called

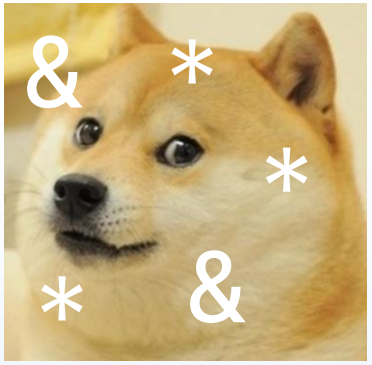
Solution: Pointer Trap

- `L03.4_pointers` on Lobster.

```
int * getAddress(int x) {
  return &x; // It's a trap!
}
void printAnInt(int anInt) {
  cout << anInt << endl;
}
int main() {
  int a = 3;
  int *ptr = getAddress(a);

  printAnInt(42);

  // should print 3, right???
  cout << *ptr << endl;
}
```

Picture

So Many * and &



- Used to specify a type…
  - * means it's a pointer

```
int *ptr;
```

  - & means it's a reference

```
int &ref;
```

- Used as an operator in an expression…
  - * means get object at an address

```
cout << *ptr << endl;
```

  - & means take address of an object

```
cout << &x << endl;
```

References vs. Pointers

References	Pointers
An alias for an object	Stores address of an object

```
int main() {
  int x = 3;
  int &y = x;
  int *z = &x;
}
```

- You can change where a pointer points.
- You cannot re-bind a reference!

What can you do with pointers?

- Work with objects *indirectly*.
  - "Simulate" reference semantics.
  - Use objects across different scopes.
  - Enable subtype polymorphism.[1]
  - Keep track of objects in dynamic memory.[1]

1 We'll look at these later in the course.

What can you do with pointers?

- Work with objects *indirectly*.
  - "Simulate" reference semantics.
  - Use objects across different scopes.
  - Enable subtype polymorphism.[1]
  - Keep track of objects in dynamic memory.[1]
  - Implement linked data structures.[1]

- **Work with *arrays* of objects.**
  - Objects in arrays have *sequential addresses*.
  - We can do *pointer arithmetic* to compute the address of the element we want.

1 We'll look at these later in the course.